2 BACKGROUND AND STATE OF THE ART

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 ... In Section 2.2 ...

TODO Describe chapter

TODO Add some words on the evergreen method of Wasm

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017 and extended by Rossberg et al. in 2018 [**?**]. It is designed to be fast, portable, self-contained and secure, and it outperforms JavaScript execution. Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2].

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecodealliance proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], making it suitable for standalone and backend execution scenarios [**?** **?**].

[0] Comp. time 2023/10/12 10:43:34

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

In Listing 2.1 and Listing 2.2, we present a C program alongside its corresponding WebAssembly binary. The C function encompasses various elements such as heap allocation, external function usage, and a function definition that includes a loop, conditional branching, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is displayed in its textual format, known as WAT[3]. The static memory declared in line 2 of Listing 2.1 is allocated within the Wasm binary's linear memory, as illustrated in line 47 of Listing 2.2. The function declared in line 5 of Listing 2.1 is converted into an imported function, as seen in line 8 of Listing 2.2. The main function, spanning lines 7 to 14 in Listing 2.1, is transcribed into a Wasm function covering lines 12 to 38 in Listing 2.2. Within this function, the translation of various C language constructs into Wasm can be observed. For instance, the `for` loop found in line 8 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The loop's breaking condition is converted into a conditional branch, as shown in line 25 of Listing 2.2.

```
1  // Some raw data
2  const int A[250];
3
4  // Imported function
5  int ftoi(float a);
6
7  int main() {
8      for(int i = 0; i < 250; i++) {
9          if (A[i] > 100)
10             return A[i] + ftoi(12.54);
11     }
12
13     return A[0];
14 }
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

[3]The WAT text format is primarily designed for human readability and for low-level manual editing.

```
1  ; WebAssembly magic bytes(\0asm) and version (1.0) ;
2  (module
3  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
4    (type (;0;) (func (param f32) (result i32)))
5    (type (;1;) (func))
6    (type (;2;) (func (result i32)))
7  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
8    (import "env" "ftoi" (func $ftoi (type 0)))
9  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
10   (@custom "name" "...")
11 ; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
12   (func $main (type 2) (result i32)
13     (local i32 i32)
14     i32.const -1000
15     local.set 0
16     block  ;label = @1;
17       loop  ;label = @2;
18         i32.const 0
19         local.get 0
20         i32.add
21         i32.load
22         local.tee 1
23         i32.const 101
24         i32.ge_s
25         br_if 1 ;@1;
26         local.get 0
27         i32.const 4
28         i32.add
29         local.tee 0
30         br_if 0 ;@2;
31       end
32       i32.const 0
33       return
34     end
35     f32.const 0x1.9147aep+3
36     call $ftoi
37     local.get 1
38     i32.add)
39 ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
40   (memory (;0;) 1)
41 ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
42   (global (;4;) i32 (i32.const 1000))
43 ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
44   (export "memory" (memory 0))
45   (export "A" (global 2))
46 ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
47   (data $data (0) "\00\00\00\00...")
48 ; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
49   (@custom "producers" "...")
50 )
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

There exist several compilers that turn source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[4], supporting a diverse set of frontend languages like C/C++,

[4] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

Rust, Go, and AssemblyScript[5]. Significantly, a study by Hilbig et al. reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a common optimization infrastructure for WebAssembly binaries. In parallel developments, the KMM framework[6] has incorporated WebAssembly as a compilation target.

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[7] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[8] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, this approach is still non-mature and faces challenges, such as the absence of JIT compilation for the "interpreted" code, making it less suitable for long-running tasks [**?**]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

2.1.2 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A Wasm binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty, i.e., they can

[5] A subset of the TypeScript language
[6] `https://kotlinlang.org/docs/wasm-overview.html`
[7] `https://github.com/bytecodealliance/javy`
[8] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

be empty. For the sake of understanding, we also annotate sections as comments in the Wasm code in Listing 2.2.

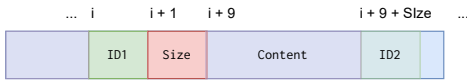


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

Due to its organization into a contiguous array of sections, a Wasm binary can be processed efficiently. For example, this structure allows compilers to speed up the compilation process through parallel parsing or just by ignoring *Custom Sections*. Additionally, the use of the LEB128[9] encoding of instructions of the *Code Section* further compacts the binary. As a result, Wasm binaries are not only fast to validate and compile but also quick to transmit over a network.

2.1.3 WebAssembly's runtime

During runtime, WebAssembly engines instantiate a WebAssembly module, which is a runtime representation of a loaded and initialized WebAssembly binary (the binary file described in Subsection 2.1.2). The key component of a module instance is its Execution Stack. The Execution Stack holds typed values, labels and control frames, with labels handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

At runtime, WebAssembly functions are closures over the runtime module instance. A function groups locals and a function body. Locals are typed variables that are local to a specific function invocation as previously discussed. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add`. As the listing also shows, instructions are annotated with a numeric type.

A WebAssembly module instance encapsulates three types of memory-related instances. First, it might include several linear memory instances, which are

[9] `https://en.wikipedia.org/wiki/LEB128`

a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are only shareable between the process that instantiates the WebAssembly module and the binary itself. Linear memories can be operated by both, the engine and the WebAssembly runtime instance. Second, global instances, which are variables with values and mutability flags, indicating whether the global can be modified or is immutable. Global variables are managed by the host engine, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them. Third, and similar to globals, locals are mutable variables that are local to a specific function instance, i.e. locals are only accessible through their index related to the executing function instance. As globals, locals are part of the managed data.

Browser engines like V8[10] and SpiderMonkey[11] are at the forefront of executing WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.2. For example, V8 just employs quick, rudimentary optimizations, such as constant folding and dead code removal, to guarantee fast readiness for a Wasm binary to execute [**?**].

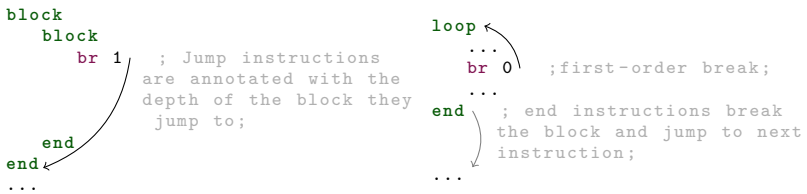
Standalone engines: Wasm has expanded beyond browser environments, largely due to the WASI[**?**]. It standardizes the interactions between host environments and WebAssembly modules through a POSIX-like interface. Wasm compilers can generate binaries that use WASI. Standalone engines can then execute these binaries in a variety of environments, including cloud, server, and IoT devices. For example, standalone engines like WASM3[12], Wasmer[13], Wasmtime[14], WAVM[15], and Sledge[**?**] have emerged to support WebAssembly and WASI. In a similar vein, Singh et al.[**?**] introduced a virtual machine for WebAssembly tailored for Arduino-based devices. Salim et al.[**?**] proposed TruffleWasm, an implementation of WebAssembly hosted on Truffle and GraalVM. Additionally, SWAM[16] stands out as WebAssembly interpreter implemented in Scala. Finally, WaVe[**?**] offers a WebAssembly interpreter featuring mechanized verification of the WebAssembly-WASI interaction with the underlying operating system.

[10] `https://chromium.googlesource.com/v8/v8.git`
[11] `https://spidermonkey.dev/`
[12] `https://github.com/wasm3/wasm3`
[13] `https://wasmer.io/`
[14] `https://github.com/bytecodealliance/wasmtime`
[15] `https://github.com/WAVM/WAVM`
[16] `https://github.com/satabin/swam`

2.1.4 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

At runtime, each break instruction is limited to jumping to one of its enclosing blocks. Excepting loop constructions, breaks jump to the end of the block, effectively continuing to the next immediate instruction. Within a loop, block ends jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design hardens the rewriting of Wasm binaries. For instance, if a block is removed, the depth of the break instructions within the block must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control flow graph to determine the enclosing block depth for each break instruction.

2.1.5 WebAssembly's binary analysis

As the WebAssembly ecosystem continues to grow, the need for robust tools to ensure its security and reliability has increased. To address this, a variety of tools have been developed that employ different strategies to identify vulnerabilities in WebAssembly programs. In the following we provide a brief overview of the most

relevant tools in this space w.r.t static and dynamic analysis, as well as specialized malware detection.

Static and dynamic analysis: Tools like Wassail[**?**], SecWasm[**?**], Wasmati[**?**], and Wasp[**?**] leverage techniques such as information flow control, code property graphs, control flow analysis, and concolic execution to detect vulnerabilities in Wasm binaries. Remarkably, VeriWasm[**?**] stands out as a static offline verifier specifically designed for native x86-64 binaries compiled from WebAssembly. In the dynamic analysis counterpart, tools like TaintAssembly[**?**], Wasabi[**?**], and Fuzzm[**?**] offer similar functionalities in vulnerability detection. Stiévenart and colleagues have introduced a dynamic approach to slice WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. Hybrid methods have also gained traction, with tools like CT-Wasm[**?**] enabling the verifiably secure implementation of cryptographic algorithms in WebAssembly.

Specialized Malware Detection: Cryptomalware have a wide presence in the web since the first days of Wasm. The main reason is that mining algorithms using CPUs moved to Wasm for obvious performance reasons [**?**]. In cryptomalware detection, tools like MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] utilize static analysis through machine learning techniques to detect browser cryptomalwares. Conversely, tools like SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**] seek the same goal with dynamic analysis techniques. Remarkably, VirusTotal[17], packaging more than 60 commercial antivirus as backboxes, detects cryptomalware in Wasm binaries.

2.1.6 WebAssembly's security

From a security standpoint, WebAssembly programs are designed without a standard library and are prohibited from direct interactions with the operating system. Instead, the host environment offers a predefined set of functions that can be imported into the WebAssembly program. It falls upon the compilers to specify which functions from the host environment will be imported by the WebAssembly application.

While WebAssembly is engineered to be deterministic, well-typed, and to adhere to a structured control flow, the ecosystem is still emerging and faces various security vulnerabilities. These vulnerabilities pose risks to both the consumers and the WebAssembly binaries themselves. Side-channel attacks, in particular, are a significant concern. For example, Genkin et al. have shown that WebAssembly can be exploited to exfiltrate data through cache timing-side

[17] `https://www.virustotal.com`

channels [?]. Similarly, research by Maisuradze and Rossow demonstrates the feasibility of speculative execution attacks on WebAssembly binaries [?]. Rokicki et al. further reveal the potential for port contention side-channel attacks on WebAssembly binaries in browsers [?]. Additionally, studies by Lehmann et al. and Stiévenart and colleagues indicate that vulnerabilities in C/C++ source code can propagate into WebAssembly binaries [? ?]. This dissertation introduces a comprehensive set of tools aimed at preemptively enhancing WebAssembly security through Software Diversification.

2.2 Software diversification

Software Diversification has been widely studied in the past decades. This section discusses its state-of-the-art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey by Baudry et al. [?], the motivation for software diversification can be separated in five categories: reusability [?], software testing [?], performance [?], fault tolerance [?] and security [?]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

TODO Work on differential testing `https://arxiv.org/pdf/2309.12167.pdf`

2.2.1 Generation of Software Variants

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[?]. This work contributes to the state of the art of Artificial Diversity, which consists of software synthesis. This thesis is founded on the work of Cohen in 1993 [?] as follows.

Cohen et al. [?] proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen and colleagues proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. [?] and the work of Jackson et al. [?], in 5 strategies.

Equivalent instructions replacement Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand

as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [**?**] and Homescu et al. [**?**] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [**?**] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [**?**] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

Instruction reordering This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [**?** **?**] proposed the random permutation of the order of variables and routines for ELF binaries.

Adding, changing, removing jumps and calls This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [**?**] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

Program memory and stack randomization This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**?** **?**] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [**?**], and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?**] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

ISA randomization and simulation This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program

can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**], and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [**?**]. Cohen [**?**] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in Subsection 2.2.1 for Cleemput et al. [**?**] and Homescu et al. [**?**], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [**?**], then this variant can be considered equivalent to the original. However, this technique is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [**?**] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be

the most used for no-branching-programs checking like basic block and peephole replacements [**?**].

Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [**?**]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

Definition 1. *Software variant* ***TODO*** *Define*

Definition 2. *Uncontrolled diversification TODO*

Definition 3. *Controlled diversification TODO*

2.2.2 Variants deployment

After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution (MVE) [**?**]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

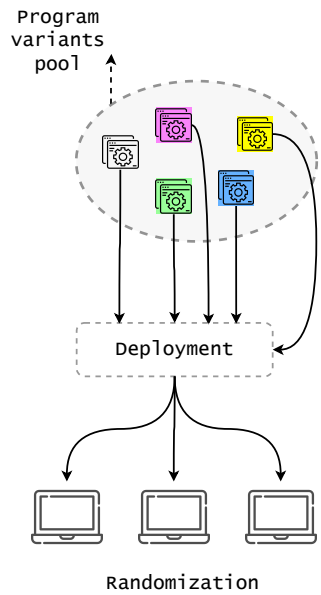
Randomization: In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [**?**] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

El-Khalil and colleagues [**?**] propose to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modify a version of GCC 4.1 to separate a conventional stack into several component parts, called multistacks. On the same topic, Aga and colleagues [**?**] propose to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different memory offsets. The Polyverse company[18] materializes randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

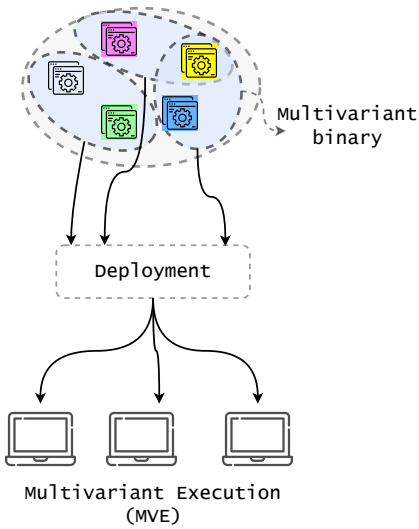
Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [**?**], create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignment of different variants to specific target clients. Similarly, Xu et al. [**?**] recompile the Linux Kernel to reduce the exposure time of persistent memory objects, increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed in one single binary (multivariant binary) [**?**]. Each multivariant binary is randomly

[18] `https://polyverse.com/`



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed. Each deployment randomly selects which multivariant binary is assigned to each host. Finally, a variant from the multivariant binary is randomly executed at runtime in the host.

Figure 2.2: *Software Diversification usages.*

deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. Figure 2.2b illustrates this scenario.

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [**?**]. Bruschi et al. [**?**] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [**?**] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [**?**], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et

al. [**?**] and Crane et al. [**?**] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling known vulnerabilities such as (JIT)ROP attacks [**?**] and power side-channels [**?**]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [**?**]. Our work contributes to both usage scenarios for Wasm.

TODO Multivariant

Definition 4. *Multivariant* ***TODO*** *Define*

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

2.2.3 Defensive Diversification

2.2.4 Offensive Diversification

2.3 Open challenges

In **??** we list the related work on Artificial Software Diversification discussed along with this chapter. The first column in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Figure 2.2.2 and Figure 2.2.2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. In the following text, we enumerate the open challenges we have found in the literature research:

1. *Software monoculture:* The same Wasm code is executed in millions of clients devices through web browser. In addition, Wasm evolves to support edge-cloud computing platforms in backend scenarios, *i.e.,* replicating the same binary along with all computing nodes in a worldwide scale. Therefore, potential vulnerabilities are spread, highlighting a monoculture phenomenon [**?**].

2. *Lack of Software Diversification for Wasm*: Software Diversification has demonstrated to provide protection for known and yet-unknown vulnerabilities. However, only one software diversity approach has been applied to the context of Wasm [**?**]. Moreover, Wasm is a novel technology

and, the adoption of defenses is still under development [**?** **?**] and has a low pace, making software diversification a possible preemptive technique. Besides, the preexisting works based on the LLVM pipeline cannot be extended to Wasm because they contribute to LLVM versions released before the inclusion of Wasm as an architecture.

3. *Lack of research on MVE for Wasm:* Wasm has a growing adoption for Edge platforms. However, MVE in a distributed setting like the Edge has been less researched. Only Voulimeneas et al. [**?**] recently proposed a multivariant execution system by parallelizing the execution of the variants in different machines for the sake of efficiency.